Algebraic Transformation of Descriptive Vector Byte-code Sequences

Larsen, Mads Ohm

Published in:
Middleware Doctoral Symposium'16 Proceedings of the Doctoral Symposium of the 17th International Middleware Conference, Trento, Italy — December 12 - 16, 2016

DOI:
10.1145/3009925.3009926

Publication date:
2016

Document version
Publisher's PDF, also known as Version of record

Algebraic Transformation of Descriptive Vector Byte-code Sequences

Mads Ohm Larsen
Niels Bohr Institute, Copenhagen University
ohm@nbi.ku.dk

ABSTRACT

Both high-productivity and high-performance are two often sought after aspects of scientific programming. Python gives the programmer high-productivity, but even with NumPy it is often not high-performant because of the GIL[1], which makes it inherently single threaded.

Bohrium intercepts NumPy calls and generates an intermediate language, Bohrium byte-code, before being compiled to OpenCL kernels. It thus grants Python/NumPy the ability to be easily run on multicore systems or GPUs, without changing the source code.

The Bohrium byte-code can be optimized, by transforming byte-code sequences into more performant ones. This way, the scientific programmer will not need to change her code to utilize special performant constructs.

CCS Concepts

•**Applied computing** → Physical sciences and engineering; •**Computing methodologies** → **Parallel computing methodologies;** Symbolic and algebraic manipulation;

Keywords

bohrium; numpy; algebraic transformation; byte-code

1. INTRODUCTION

Python is an interpreted, high-level, general purpose programming language, which enables high-productivity and readability. For scientific applications Python programmers utilize NumPy [4], which has become the de-facto standard for vectorized code[2] for Python.

Bohrium [2, 3] extends NumPy, by allowing the code to be run on multicore CPUs, clusters, or GPUs, without interfering with the high-productivity aspect of Python. The programmer only has to change the `import` from `numpy` to `bohrium`, and the Bohrium runtime will take over, still utilizing all NumPy calls internally.

The byte-code language operates on tensors[3] of varying size and shape. Every time the programmer invokes a NumPy method, Bohrium intercepts it and, if possible, does the computation on e.g. the GPU instead.

Since Bohrium has multiple front-ends, e.g. Python, CIL, C++, this project will focus on optimizing the intermediate language, that the Bohrium runtime generates. An optimization would be transforming byte-code sequences that can be rewritten into more efficient code, thus allowing the programmer high-productivity as well as fast running code.

2. ALGEBRAIC TRANSFORMATIONS

A transformation can be thought of as a rewriting of elements from one set to another.

An example of such a transformation could be to realize that

$$x^n \mapsto \underbrace{x \cdot x \cdot \cdots \cdot x}_{n} = \prod^{n} x \quad \text{if } n \in \mathbb{N} \tag{1}$$

We can thus exchange the power-function for a series of multiplications or vice versa. This also holds true for tensors, such that $\vec{x}^{10} = \prod^{10} \vec{x}$.

Another example of a transformation, one that requires a more context-aware transformation, could be transforming a solution of

$$\vec{A}x = \vec{B}$$

, where $\vec{A}$ and $\vec{B}$ are tensors. Usually we first have to find the inverse $\vec{A}^{-1}$ tensor to solve for x.

$$\begin{aligned} \vec{A}x &= \vec{B} && \Leftrightarrow \\ \vec{A}^{-1}\vec{A}x &= \vec{A}^{-1}\vec{B} && \Leftrightarrow \\ x &= \vec{A}^{-1}\vec{B} \end{aligned} \tag{2}$$

Instead one could do a LU-factorization [1] of the same problem, which would usually be faster to compute. Note that this is of course only faster, if we do not use the $\vec{A}^{-1}$ tensor for anything else in our computations.

The transformations can thus be small loop-fusion-like contractions of byte-codes, or it can detect the semantic meaning of the code, thus being more specialized and context-aware.

[1] Global Interpreter Lock
[2] Also known as array-programming.
[3] Multi-dimensional matrices.

Middleware Doctoral Symposium '16, December 12-16 2016, Trento, Italy

ISBN 978-1-4503-4665-8/16/12...$15.00

DOI: http://dx.doi.org/10.1145/3009925.3009926

3. BOHIRUM BYTE-CODE

Even though Python is an interpreted language, with Bohrium we actually get a JIT-compiled intermediate language. We can thus manipulate the byte-code before executing it.

In this byte-code language a single line encapsulates one byte-code. A byte-code consists of an op-code, e.g. `BH_ADD`, a result register, and up to two parameter registers or constants.

In Listing 1 we have a Python program adding three ones together in a one-dimensional vector of size 10.

```
import bohrium as np
a = np.zeros(10)
a += 1
a += 1
a += 1
print a
```

Listing 1: Adding three ones in Python.

In Listing 2 the same program is shown in the Bohrium byte-code langauge.

```
BH_IDENTITY a0[0:10:1] 0
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_ADD a0[0:10:1] a0[0:10:1] 1
BH_SYNC a0[0:10:1]
```

Listing 2: Adding three ones with Bohrium.

Here, `a0` is a vector register where our view is always from `0` to `10` with a step of `1`. In further listings I assume the view is the same for all registers, and thus will not write it out.

What this means is that we three times add 1 to all elements of our `a0` tensor's full view and store it back into the full view.

3.1 Transforming the Byte-code

In the code example in Listing 2, we see each of the three additions being their own byte-code. In reality our `a0` tensor could be very large, so large in fact that adding one to each element would take a long time. Instead the constants of the three byte-codes can be merged into one by simply adding them together. After transforming the byte-code sequence, we thus end up with only one `BH_ADD` op-code adding 3 to each element, as shown in Listing 3.

```
BH_IDENTITY a0 0
BH_ADD a0 a0 3
BH_SYNC a0
```

Listing 3: Optimized adding three ones with Bohrium.

By transforming we can also generate more byte-codes. In the power-to-multiplication example we start with one byte-code, `BH_POWER`, and can potentially end up with many. Here it is important to know, that we usually only have access to the origin and result tensors, since copying data to create temporary tensors would be time consuming for large tensors. For the power example from (1), we can get better performance, still only using `BH_MULTIPLY`, by utilizing the result tensor multiple times, instead of only seeing it as the end-result. To do this, we can realize that

$$\begin{aligned} x^{10} &= x^8 \cdot x \cdot x \\ &= x^4 \cdot x^4 \cdot x \cdot x \\ &= x^2 \cdot x^2 \cdot x^2 \cdot x^2 \cdot x \cdot x \end{aligned}$$

If we thus first calculate $x^2 = x \cdot x$ and store that in our result tensor `a1`, we can then multiply it with itself to get x^4. Multiplying this with itself grants us x^8 and then multiplying this with x twice gives us the result x^{10}.

In Listing 4 we have calculated x^{10} by multiplying first our origin tensor by x and then 9 more times multiplying the result tensor with x.

```
BH_IDENTITY a0 ...      # initialize the tensor, x
BH_MULTIPLY a1 a0 a0    # x^2
... (x 7)               # x^3..x^9
BH_MULTIPLY a1 a1 a0    # x^10
BH_SYNC a1
```

Listing 4: $\vec{x}$ to the power of ten, using nine `BH_MULTIPLY`s.

We could actually do better. Since we own the result tensor, we are allowed to use it as we see fit. In Listing 5 a better x^{10} is shown.

```
BH_IDENTITY a0 ...      # x
BH_MULTIPLY a1 a0 a0    # x^2
BH_MULTIPLY a1 a1 a1    # x^4
BH_MULTIPLY a1 a1 a1    # x^8
BH_MULTIPLY a1 a1 a0    # x^9
BH_MULTIPLY a1 a1 a0    # x^10
BH_SYNC a1
```

Listing 5: $\vec{x}$ to the power of ten, using just five `BH_MULTIPLY`s.

4. CONCLUSIONS

Some of these rudimentary transformations have already been implemented into the Bohrium runtime system. Bohrium already supports merging integer addition, by adding the constants, before adding it to the actual tensors. It also does power expansion by default, since benchmarks have shown, that for values close to a power of 2, multiplying multiple times is faster than doing an actual `BH_POWER`.

A further study of real examples, such as (2), from imaging software and benchmark suites is planned. If resulting sequences is found to be too slow, a study on how to make them faster will be made.

5. ACKNOWLEDGMENTS

This work is part of the CINEMA project and financial support from CINEMA: the allianCe for ImagiNg of Energy MAterials, DSF-grant no. 1305-00032B under The Danish Council for Strategic Research is gratefully acknowledged.

Thanks goes to Sarah Haas for her Writer Development course as well as tools on how to write in protected environments.

6. REFERENCES

[1] J. R. Brunch and J. Hopcroft. Triangular factorization and inversion by fast matrix multiplication, 1974.

[2] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: a Virtual Machine Approach to Portable Parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[3] M. O. Larsen, K. Skovhede, M. R. B. Kristensen, and B. Vinter. Current Status and Directions for the Bohrium Runtime System. In *Compilers for Parallel Computing 2016*, 2016.

[4] T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.